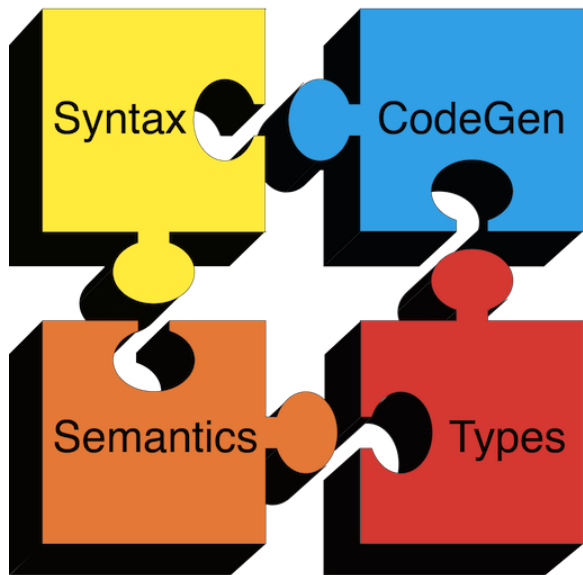




Compilers

Structure of a Compiler

1. Lexical Analysis
2. Parsing
3. Semantic Analysis
4. Optimization
5. Code Generation

- First step: recognize words.
 - Smallest unit above letters

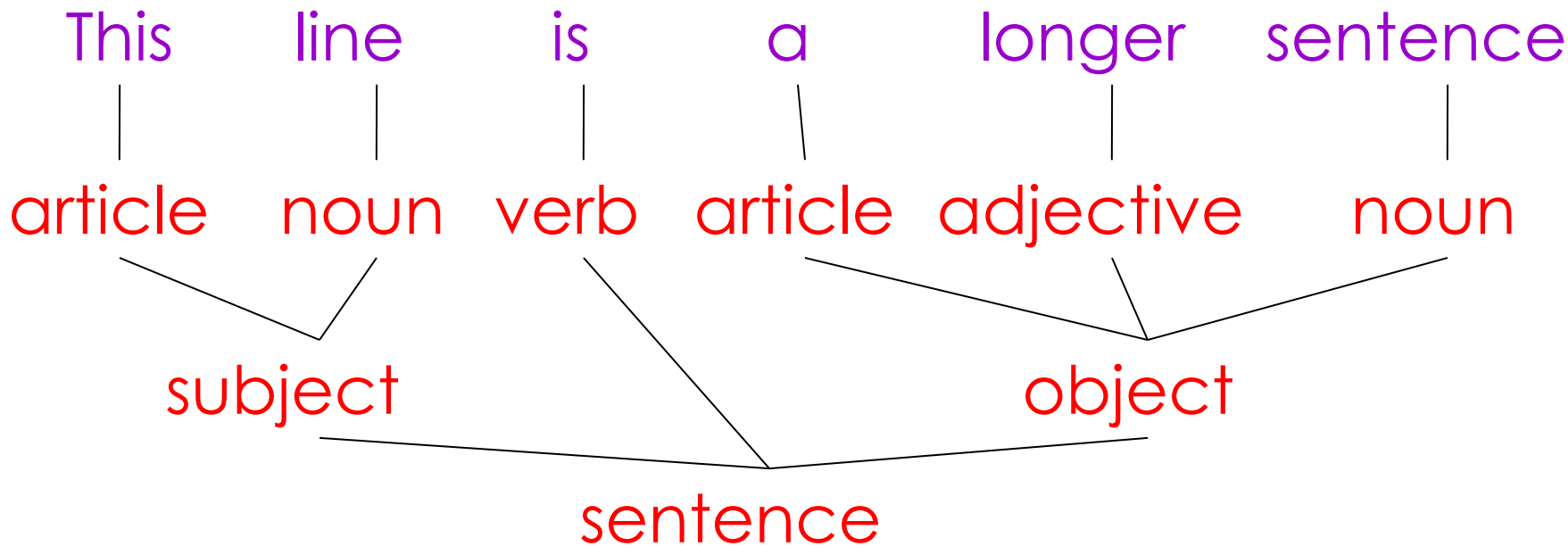
This is a sentence.

ist his ase nte nce

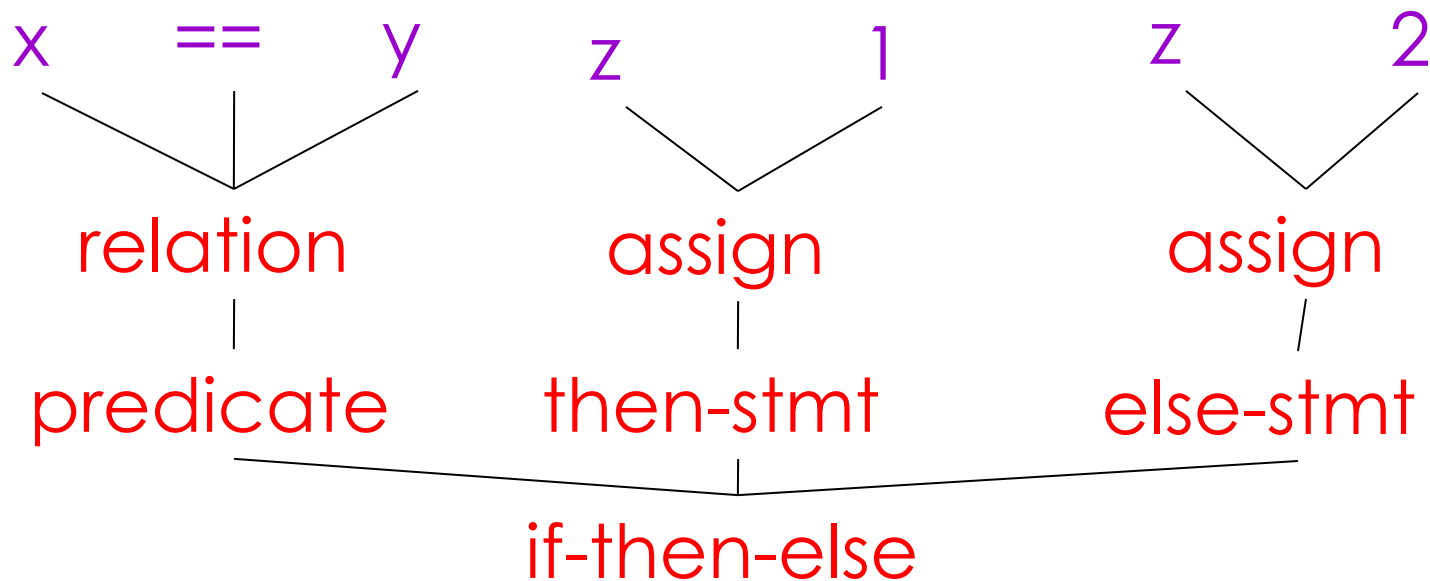
- Lexical analysis divides program text into “words” or “tokens”

if x == y then z = 1; else z = 2;

- Once words are understood, the next step is to understand sentence structure
- Parsing = Diagramming Sentences
 - The diagram is a tree



if x == y then z = 1; else z = 2;



- Once sentence structure is understood, we can try to understand “meaning”
 - This is hard!
- Compilers perform limited semantic analysis to catch inconsistencies

- Example:

Jack said Jerry left his assignment at home.

- Even worse:

Jack said Jack left his assignment at home?

- Programming languages define strict rules to avoid such ambiguities

```
{  
  int Jack = 3;  
  {  
    int Jack = 4;  
    cout << Jack;  
  }  
}
```

- Compilers perform many semantic checks besides variable bindings

- Example:

Jack left her homework at home.

- A “type mismatch” between her and Jack; we know they are different people

- Optimization has no strong counterpart in English
 - But a little bit like editing
- Automatically modify programs so that they
 - Run faster
 - Use less memory

$X = Y * 0$ is the same as $X = 0$

- Produces assembly code (usually)
- A translation into another language
 - Analogous to human translation

- The overall structure of almost every compiler adheres to our outline
- The proportions have changed since FORTRAN